面向Cassandra数据库的高效动态数据管理机制

王博千 于 齐 刘 辛 沈 立 王志英 陈 微
(国防科学技术大学计算机学院 长沙 410073)

摘 要 Cassandra数据库是当前通用的数据库之一,同时也被Apache列为重点发展的顶级项目。针对Cassandra分布式数据库系统而言,大量的写请求会造成过多分散的SStable结构以及过高的数据冗余度,进而造成用户读取请求响应速度下降。该问题可以通过系统自动触发的局部数据合并机制或人为干预的整体数据合并机制来解决。然而,不合时机的系统自动局部合并过程会严重降低用户正在执行的读取操作的性能,而过长时间的人为整体数据合并过程又会长时间地占用系统资源,严重制约系统的整体性能。针对此问题,提出了一种面向Cassandra数据库的动态数据管理机制。首先,实时监测系统环境,将数据按照写入时间和大小进行分层分级管理,对合并的时机、参与合并的文件及合并过程分别制定相应的执行策略;其次,通过特定优化手段减少数据的合并时间,以降低合并过程对系统性能的影响。测试结果表明,该管理机制优化了Cassandra数据库的合并过程,提升了系统整体的读取响应速度。

关键词 Cassandra数据库,动态数据管理,合并策略,读取响应速度
中图法分类号 TP311.133.1 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2016.7.036

Efficient and Dynamic Data Management System for Cassandra Database

WANG Bo-qian YU Qi LIU Xin SHEN Li WANG Zhi-ying CHEN Wei
(School of Computer, National University of Defense Technology, Changsha 410073, China)

Abstract Cassandra is one of the universal databases, and it's also specified as the top level project by the Apache. For the Cassandra distributed database system, a large number of write requests will cause excessive and dispersed SStable structures and high data redundancy, causing low efficiency to the user read requests. This problem can be solved by the local data consolidation mechanism triggered automatically by the system or by the overall data consolidation mechanism triggered by the human intervention. However, on one hand, the irrational timing automatical partial merger process will seriously reduce the performance of the read operation requested by the user; on the other hand, the long-time human overall data consolidation process will occupy a large number of system resources, which will severely restrict the overall performance of the corresponding system. To solve this problem, we presented an efficient and dynamic management mechanism. Firstly, appropriate implementation strategies are developed to the time of the merger, the file involved in the merger and the merge process by monitoring system environment and managing the data according to the time and size. Secondly, the impact of the consolidation process on system performance is reduced by reducing the data combination time through specific optimization methods. The final result shows that this data management system optimizes the Cassandra database consolidation process and ultimately enhances the response speed for the read request.

Keywords Cassandra database, Dynamic data management, Consolidation strategy, Response speed for the read request

1 引言

随着互联网的发展,人们已进入数据爆炸的时代,越来越多的数据将会被存储和查询。由此,大数据的概念便进入了人们的视野。大数据指的是所涉及处理的资料数据规模巨大,以至于一些传统的应用软件架构以及硬件体系架构无法与之相适应。大数据应用过大的数据量增加了数据的存储压力,同时也给传统的数据处理分析技术带来了巨大的挑战,其所带来的主要问题是系统的数据处理能力低下、用户的等待时间变长。为了解决此类问题,在硬件上可以针对大数据应用请求数据关联度低、并行性好、计算过程简单的特点,通过化简处理器核结构、增加处理器核数、减少多级存储结构的方法来加速大数据应用[1-3];在软件上可以通过动态的数据管理机制及增加大数据应用对硬件以及空闲时间的利用机制来优化效果[4]。本文通过软件动态数据管理机制对经典的大数据应用——Cassandra数据库系统进行了优化,改进了其数据的

到稿日期:2015-05-18 返修日期:2015-08-31 本文受国家自然科学基金项目(61472431,61202121),教育部高等学校博士点新教师基金项目(20114307120013)资助。

王博千(1990—),男,硕士,主要研究方向为系统结构,E-mail:wangboqian1990@gmail.com;于 齐(1990—),男,硕士,主要研究方向为系统结构;刘 辛(1990—),女,硕士,主要研究方向为系统结构;沈 立(1976—),男,副教授,主要研究方向为系统结构;王志英(1956—),男,教授,主要研究方向为系统结构、信息安全;陈 微(1982—),女,讲师,主要研究方向为高性能微处理器设计。

管理过程，并最终获得了性能的大幅度提升。

Cassandra 数据库是一个典型的分布式 NoSQL 数据库系统，由 Facebook 开发，以 Amazon 专有的完全分布式数据库 Dynamo 为基础，结合了 Google BigTable 的列存储类型，在很多方面都可以称之为 Dynamo 2.0；而后其又成为 Apache 顶级项目，如今正跻身于数据库排行榜的 Top10 行列之中，由于其良好的可扩放性，被 Digg、Twitter 等知名 Web2.0 网站所采纳[5,6]。

Cassandra 数据库主要采用 key-value 对的存储模型，在内存中采用 MemTable 数据模型，先将对磁盘随机写、修改和删除的数据缓存到内存中，按照键值的大小顺序存放。当内存中的 Memtable 达到一定大小后再将其顺序写入到磁盘上，在磁盘上则采用 SStable 数据模型[7]。这种机制在一定程度上保证了 Cassandra 数据库的写入、删除和修改操作的性能，但是也造成了数据的大量冗余，对数据库的数据读取操作性能造成了严重影响。同时过多分散的 SStable 架构产生了大量合并操作，占用了过多的硬件资源和数据带宽，将会严重影响数据库的性能。

基于上述问题，本文采用一种动态的数据管理机制，一方面，加速 SStable 的合并过程，减少因合并过程而占用大量硬件资源的时间；另一方面，通过监测系统整体性能，动态化地采用相应的数据合并策略减少数据冗余，提升用户请求响应速度。

2 Cassandra 数据库行为分析

本节将详细描述 Cassandra 数据库在单节点的数据读写流程，通过对数据处理流程及实验测试结果进行分析，得出制约数据库性能的瓶颈因素。

2.1 数据处理流程分析

Cassandra 数据库在单节点上的运行流程如图 1 所示。

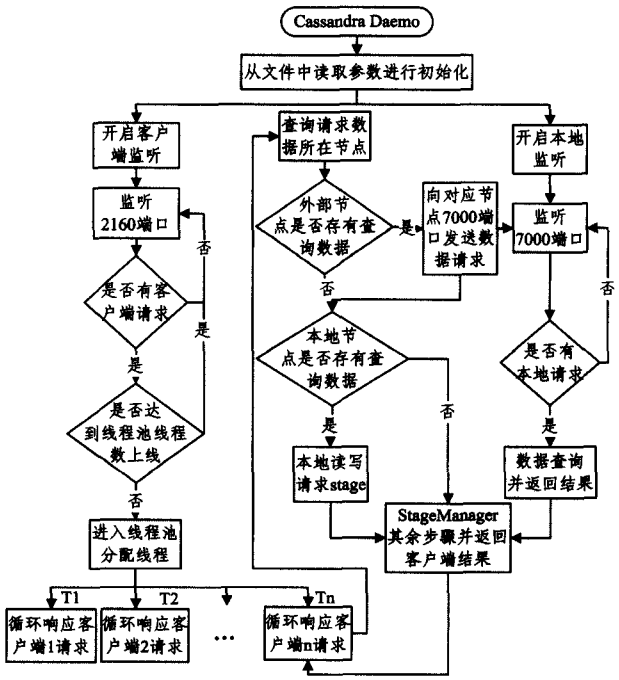

图 1 Cassandra 数据库单节点运行流程

其主要是通过监听 2160 端口来响应用户的数据请求，接到用户请求后便从线程池中创建线程来专门处理该请求，之后通过查询获得请求数据所在的节点，并向其 7000 端口发送数据请求。在每个存有该数据的服务器本地，实现其底层的读写请求过程，并通过 StageManager 管理机制来维护服务器集群稳定、节点通讯和数据一致性。本文将重点讨论单节点上 Cassandra 数据库的读写机制及流程，通过优化单节点性能来实现服务器集群整体性能的提升。写请求响应过程具体如图 2 所示。

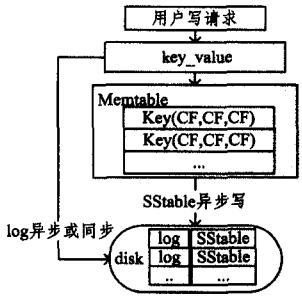

图 2 Cassandra 数据库写入请求流程

在写入过程中，首先写入 log 记录文件，保证断电后可以恢复系统数据，之后对于写入的每个 key-value 对，按照 key 值的大小顺序依次写入内存的 Memtable 结构中，当 Memtable 结构达到一定大小后，通过后台将整个 Memtable 结构转化为 SStable 结构写入磁盘。同理，针对修改和更新请求，Cassandra 将其化为统一的写请求过程，通过添加修改、更新记录，最终在进行 SStable 合并时实现数据的真正更新和删除。该过程将数据的随机写操作变成了顺序写操作，通过内存缓存机制，减少了对磁盘的直接写入以及更改、删除，极大地提升了数据库响应写入、删除和更新请求的性能。

相比之下，其数据读取操作更为复杂。针对同一个数据，其可能存在的新老副本被分别存储在内存中的 Memtable 结构和磁盘上的多个 SStable 结构中，因此对于数据的查询操作，就要求读取其数据的所有副本，通过合并机制查找最新记录并返回给用户。具体过程如图 3 所示。

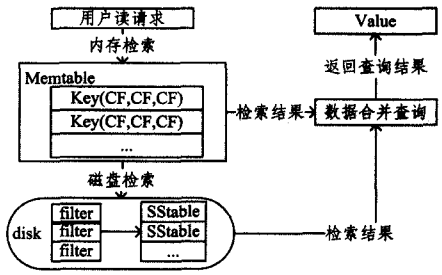

图 3 Cassandra 数据库读取请求流程

读取过程中，首先根据 row_key 值在 Memtable 中查找记录结果，之后通过 row_key 值在众多的 SStable 结构中定位其所存在的位置，并将相应的 SStable 结构读入内存，最终合并读取的 Memtable 和 SStable 记录并通过 row_key 值查询获取最新的数据结果返回给用户。在这个过程中存在着大量的磁盘读取操作，第一次是为了定位对应的 key 值所在 SStable 结构的位置，要查询磁盘文件 bloom filter；第二次是读取磁盘上特定的某几个具有该记录的 SStable 结构[8,9]。虽然在 Cassandra 内部试图通过 Row 缓存和 Key 缓存来减少磁盘访问，但是随着数据量的增加，它们都暴露出了相应的弊端：Row 缓存数据量过大，造成了内存的大量占用；Key 缓存虽然占据空间较小，但是仍然无法避免查询过程中第二次

磁盘读写操作。

因此，在现有的 Cassandra 数据库机制中，并没有实现对读操作的显著优化，因此在整个系统流程中，读取操作便成为了系统的主要瓶颈。

2.2 性能瓶颈测试

测试实验的软、硬件环境如表 1 所列。

表 1 系统测试的软、硬件环境

	硬件测试环境
1	CPU Intel(R) Core(TM) i7-2600 3.40GHz，四核、八线程
2	L1 cache 256k * 4，L2 cache 1M * 4，LLC cache 8M * 1
3	Memory DDR3 1600MHz
	软件测试环境
1	Ubuntu Desktop 14.04，内核版本 3.13.0
2	Java 版本 1.8.0_20
3	Cassandra 2.1.1，YCSB 0.1.4

针对程序流程分析结果，下面对标准测试程序 YCSB[10] 在相同的软硬件环境下进行测试，读、写以及删除、更新操作的性能对比结果如表 2 所列。此为在总数据量为 10GB，只包含 1 个 SStable 的环境下的测试结果，在对读取过程测试前已经充分填充了页高速缓存，使读性能达到最佳。

表 2 读、写、删除、更新操作的性能对比

操作	请求数量	用时(s)	执行速度(ops/sec)
数据读取	1000000	4400	227
数据更新	1000000	27	37037

可以看出，数据的读取和更新执行速度相差在 100 倍左右。Cassandra 数据库的删除、更新以及写操作的执行速度基本上是保持在同一数量级的，这与数据库本身的操作实现机制有关，这 3 个操作本质上都可以看作对数据库的写操作，主要是对内存的顺序写操作，同时掩盖了后台将数据写入磁盘的操作。但对于读操作而言，由于其中存在大量的磁盘随机读操作，因此相比之下其效率便较为低下。下面将通过进一步测试分析，得出制约数据库读取性能的具体因素。

首先测试页高速缓存对 Cassandra 读取操作性能的影响。页高速缓存是 Linux 内核实现的一种主要磁盘缓存，它主要用来减少对磁盘的 IO 操作，具体讲，就是通过把磁盘中的数据缓存到物理内存中，把对磁盘的访问变为对物理内存的访问。一般情况下，在 Linux 操作系统中，页高速缓存占用系统绝大多数内存，但会主动释放，不会影响系统和应用的内存占用，更不会影响系统性能。在 8GB 内存总容量下，设置 JVM 虚拟机内存 2GB，此时页高速缓存能达到5.5GB左右。具体测试结果如图 4 所示[11]。

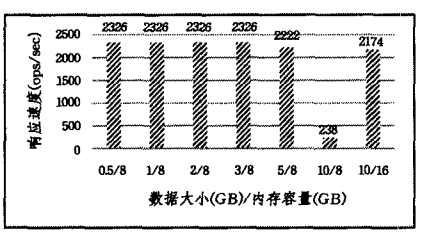

图 4 页高速缓存相对于数据大小情况对应用性能的影响

从图 4 可以看出，当数据量大小可以完全存入页高速缓存时，其用户的数据响应速度都能达到一个较高的水平；而当数据量超过页高速缓存大小时，便会造成性能的严重下降，此时会产生大量的磁盘读和内存块的替换操作，成为影响系统性能的一个瓶颈因素。

其次，Cassandra 的读写过程会产生大量的 SStable 结构，这样在进行读取操作时会查询大量的 SStable 结构，同时需要将其载入内存后再进行合并查询[12]。过多的 SStable 结构对读性能的影响如图 5 所示。

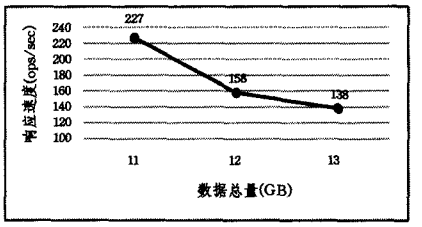

图 5 过多 SStable 结构对读取请求性能的影响

测试的初始环境为 1 个大小为 11GB 的 SStable 文件，第 2、3 次测试为执行更新操作后的结果，每次操作后增加 17 个大小为 70MB 左右的 SStable 文件，由于是更新操作，有效数据总量不变。

从图 5 可以得出结论：过多的 SStable 结构造成了读取过程中的大量磁盘随机读操作，严重影响了用户的读取访问请求，因此有必要在一定条件下对 SStable 进行合并操作。Cassandra 应用提供了两套合并机制：1) Major compaction，主要由管理员进行手动操作，将所有的 SStable 进行合并，但是这一过程耗时巨大；2) Minor compaction，当磁盘上的 SStable 符合一定数量时，由系统自动执行。但是合并过程中将会占据几乎所有的数据带宽，这样将会对用户请求的响应造成严重的影响。具体如图 6 所示。

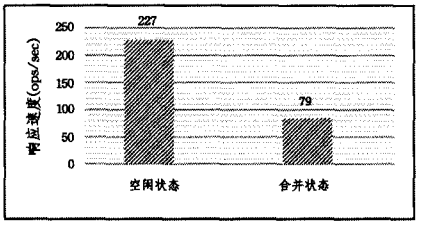

图 6 系统合并过程对读写请求性能的影响

通过测试结果可以发现，数据的合并操作将会导致用户的读取请求性能严重下降，因此过多冗余的 SStable 结构和过长的随机合并操作造成了系统的另一个性能瓶颈。

3 优化策略

通过上述性能测试和分析，不难发现，造成用户读请求性能瓶颈的主要原因如下。

过多的 SStable 造成了大量的数据冗余和磁盘随机读写操作；过大的数据尤其是数据冗余造成了页高速缓存不足或是利用不充分；不合理的 SStable 合并时机以及过长的合并时间占用大量系统资源，严重影响了用户请求。

针对上述性能瓶颈，本节提出了以下两个性能优化策略：

(1)在特定的系统状态下按照合并策略实现自定义 SStable 合并，以减少数据冗余。

Cassandra 应用本身自带两套合并机制：1) Major compaction，主要由管理员进行手动操作，将所有的 SStable 进行合并，但是这一过程耗时巨大，占用的资源极多，合并过程将会对用户的请求产生严重的影响；2) Minor compaction，当磁盘上的 SStable 达到一定数量时，由系统强制执行，于是合并的时机无法人为干预，可能会遇上用户请求的高峰期，对用户

请求的响应速率产生严重影响，同时选择参与合并的 SStable 不一定最优。

于是提出一种改进的合并策略，该策略主要通过实时监测服务器状态，确定当前是否正在进行大量的用户读写操作。若在一定时间内服务器的处理器和数据带宽都保持在一个较低的水平，则认定为是用户请求的低谷期，开始进行 SStable 合并。合并过程主要是根据服务器的可用硬件资源多少将所有 SStable 划分为多个等级，较小的 SStable 结构存放于较低的等级，然后由下至上依次在不同的等级进行数据合并操作。

由于系统的实现机制所限，必然会产生越来越多的 SStable 结构，因此必然要经历大量的合并操作。对于繁忙的在线系统而言，总会存在用户请求的高峰和低谷时期，同时各个服务器之间的繁忙程度也会存在一定的差异，因此利用服务器的响应低谷时期进行合理的合并操作就显得更加合理可行。相比于系统本身的 Major compaction 而言，该策略更加充分地利用了系统琐碎的空闲时间，减少了由于合并操作对于系统资源大规模长时间的占用。而相比于 Minor compaction 而言，本策略由于实时监测系统的状态，因此能使得合并时机的选择更加合理。

此外，为了避免长时间的合并操作对用户请求造成的过长延迟，在合并的过程中将会实时监测用户请求，当过多的用户请求造成过高的 CPU 和带宽使用率后，会自动终止合并操作，进行数据回滚。

系统的合并机制为：将待合并的 SStable 文件加载到内存中，并将合并过程中产生的临时文件实时刷新到磁盘上，当合并过程结束时，将临时文件改变为永久性文件，并同时删除参与合并的原始 SStable 文件。这样就由系统本身的实现机制所限定，回滚操作只能直接将合并过程中产生的临时文件直接删除，同时继续保留原始数据，即该回滚操作会将所有的数据回滚到合并之前的状态。因此为了减少回滚操作对整体性能产生的过大影响，设定在特定某个等级的 SStable 集合中按照文件写入时间的先后顺序进行排序，优先合并新写入的 SStable 结构，并且将其数据总量控制在一定的范围内，尤其是不能超过页高速缓存的总容量的一半。

这样通过控制一次合并的总量，既可以减少由于系统负荷增大时产生的回滚操作对系统性能所造成的影响，又能将所有数据一次性加载到内存中进行合并，缩短了合并过程的时间，进一步增加了对于用户请求低谷时期的利用率以及合并机制的可行性。

(2)通过预取和多线程机制减少 SStable 合并时间。

为了减少合并过程对用户的影响，采用一些策略来减少合并过程的时间。首先通过数据预取操作将要合并的数据提前载入到页高速缓存中。由于是数据的整体载入，其效率会高于直接进行合并的操作。其次，由于数据合并不会占用太多的 CPU 资源，因此可以采用多线程机制，将已载入到内存的数据进行并发合并，加速合并过程。通过此机制，既可以减少合并过程的总时间，又可以缩短单次合并过程的时间，增加系统空闲时间的可利用率，减少回滚操作的出现，使得利用系统较为空闲的时间进行合并的策略更加合理可行。

具体自定义的数据合并策略实现机制的伪代码如下。

函数伪代码描述：

```
while(true){//循环检测系统环境并执行相应操作
  GetSystemInfo();//获得系统环境参数
  If 系统环境达到特定条件 then
/*读取文件创建时间及大小信息，根据文件大小范围列入不同 list 中，内部按创建时间逆序排序*/
    listinfo=ReadFileInfo(userspace);
    list []=SortByFileInfo(listinfo);
/*对第一个不为空的 list 执行如下操作*/
/*若文件数为 1，则转存至下一个 list 中*/
    ConsolidationFileList=GetFile(list);//从 list 中获取一定数量或一定大小总量的文件
    pthread(ConsolidationFileList).start();{ //线程部分
      Prefetch(ConsolidationFileList);//预取文件
      UserDefinedCompaction(ConsolidationFileList);//多线程并行执行用户定义的特定文件合并
      InfoMainThread();//通知主线程合并过程结束
    }
/*主线程执行部分*/
    while(true){ //循环检测系统状态及线程执行情况
      GetSystemInfo();
      If 请求过多导致系统达到特定状态 then
        Interrupt(pthread);
        break;
      End if
      DetectPthreadRunInfo();
      If 线程执行完毕 then
        break;
      End if
      Sleep(time);
    }
    End if
    Sleep(time);}
```

由上述的伪代码可以看出，主要通过主线程监视服务器的性能参数，决定何时开始以及选取哪些文件执行合并操作。当达到特定条件后开启新的线程来执行预取和多线程合并，此时主线程仍然检测系统环境，判断执行过程中是否出现了大量的用户请求，若如此则及时终止合并过程，以保证用户请求的性能不受影响；若合并操作正常终止，则重新循环执行上述过程。

4 性能测试和分析

4.1 合并时间

首先测试通过数据预取手段加速数据合并的效果。针对要进行合并的数据，先将其加载到页高速缓存之中，当所有待合并数据预取完毕后再执行数据合并操作。具体结果如图 7 所示。

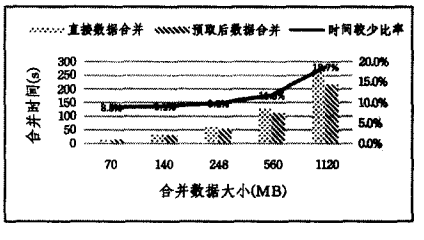

图 7 预取合并策略对合并过程的影响

图 7 中左边柱状图为采用直接合并策略花费的时间，右边柱状图为采用先预取再合并策略花费的时间，可以发现，针

对同样大小的数据合并总量而言，预取合并策略的时间开销要小于直接合并的时间开销，并且随着合并数据总量的增加，其时间减少的优化效果也越来越明显。

其次在数据预取的基础上考虑多线程优化机制对合并优化的效果。这里先考虑多线程机制对直接合并操作的影响，如表 3 所列。

表 3　多线程直接合并策略对合并过程的影响

直接合并(MB)	单线程(s)	双线程(s)	性能提升(%)
70 * 8	66	129	2.27
560 * 2	126	248	1.59

可以看出，对于直接合并而言，多线程机制并不能明显减少合并时间。这可能是由以下两方面之一引起的：

1)单线程数据预取占用了过多的数据带宽，从而导致多线程合并过程在数据预取过程中产生了堵塞，进而导致合并性能提升不明显。

2)数据内存合并过程占用了大量的系统资源，如内存空间和 CPU 功能单元，从而导致数据在内存合并过程中产生堵塞，导致合并性能提升不明显。

针对第一种情况，分别对单线程和多线程预取数据进行测试，其中要保证每个线程读取的数据量一致，测试结果如表 4 所列。

表 4　多线程预取策略对预取性能影响

数据大小 (MB)	运行时间(s)(按最慢的线程计算)			
	单线程	双线程	三线程	四线程
560	11	22(22)		
1120	24	49(48)	75(72)	102(96)

表 4 中括号内数据为在保证预读相同数据量的前提下，采用单线程串行执行的理论时间。可以看出，多线程预读数据的时间是随着线程数量成倍增加的，由此得出结论，多线程预读机制并不能提升合并的效果。

针对第二种情况，先将同样大小的数据预取到页高速缓存中，之后分别采用单线程和多线程合并机制进行测试，结果如表 5 所列。

表 5　多线程合并策略对合并过程的影响

合并数据大小(MB)	单线程(s)	双线程(s)	性能提升(%)
280 * 2	48	82(96)	14.58
560 * 2	94	162(188)	13.83
1120 * 2	176	418(352)	−18.75

表 5 中括号内数据为合并相同数据量的前提下，采用单线程串行执行的理论时间。可以看出，前两次测试采用双线程机制都能获得合并性能的提升。针对第 3 次测试，由于其双线程合并过程的总数据量达到了 4GB，其合并的结果文件也会缓存到页高速缓存之中，这样将近 8GB 的数据量远远超过了测试环境中的页高速缓存容量上限，因此会导致大量的磁盘读写和内存数据替换过程，从而严重影响数据的合并过程。

由此得出结论，将数据的合并过程拆分为数据预取与数据内存合并两个过程可以加速优化数据的合并过程。其中数据预取过程采用单线程机制实现，数据内存过程采用多线程并发加速，同时要保证合并的原始数据和生成数据总量不超过系统内存中页高速缓存的大小。

4.2　读取响应速度

最终优化的测试结果如图 8 所示，初始化的测试数据集为 1 个大小为 11GB 的 SStable 结构，之后的每次测试都是先更新 4GB 的数据，再按上述管理策略合并后进行测试的数据。

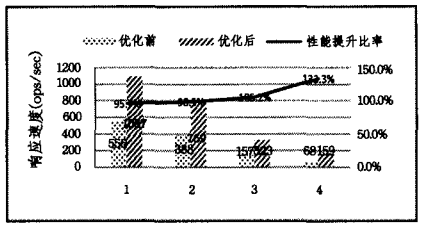

图 8　优化前后用户读取请求响应的性能对比

由图 8 可以看出，优化后系统的读取响应性能相较优化前而言有了很大的提升，最好情况下达到了 133%的性能提升比。同时测试结果表明，由于去除了数据冗余，最后数据总量由理论上的 23GB 变成了 13GB，冗余数据由的 12GB 缩小到了 2GB，减少了 83%。这样一方面减少了过多 SStable 的读取和内存合并操作，另一方面减少了数据冗余，增加了页高速缓存的利用率，实现了性能的显著提升。

4.3　合并过程对系统性能的影响

合并过程占用了大量的系统资源，尤其是系统带宽和内存资源，因此必然会在一定程度上影响系统的整体性能，通过实验测试，在满负荷的用户读写请求下，具体结果如图 9、图 10 所示。

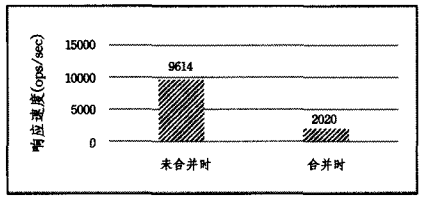

图 9　合并操作对用户写请求的影响

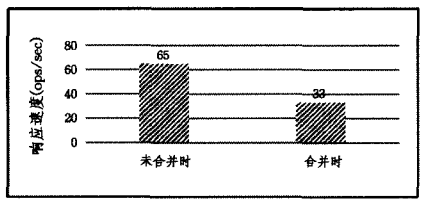

图 10　合并操作对用户读请求的影响

从图中可以看出，合并过程对读、写过程都产生了一定的影响，在读、写满负载的条件下，读操作速率下降了近 5 倍，写操作速率下降了近 2 倍。但是合并操作是不可避免的，关键在于选择合并的时机以及缩短合并过程。对于系统本身的合并机制而言，无法做到对合并时机以及合并时间的控制，而本文的合并策略则从选取合适的合并时机以及缩短合并过程两方面进行了优化，选择在用户请求的低谷期、读写负载相对较低的时期进行合并操作，可以有效减少由于合并过程占用系统资源而对用户的读写响应速率所产生的影响，同时也缩短了合并时间，减小了合并过程对系统性能的影响，进一步增加了该策略的效果和实用性。

结束语　通过对 Cassandra 数据库的流程分析以及性能测试找出了制约其系统性能的主要瓶颈。针对其设计的不足，首先通过一定的机制检测服务器状态并适时采用相应的策略进行特定数据的合并操作，其次通过预取和并发机制减

少数据的合并时间，减少合并过程对系统性能的影响。测试结果表明，通过以上策略可以明显减少数据的合并时间，提高Cassandra服务器的读取响应性能。

参考文献

[1] Ferdman M, Adileh A, Kocberber O, et al. Clearing the clouds: a study of emerging scale-out workloads on modern hardware[J]. ACM SIGARCH Computer Architecture News, 2012, 40(1): 37-48

[2] Lotfi-Kamran P, Grot B, Ferdman M, et al. Scale-out processors [J]. IEEE Computer Society ACM SIGARCH Computer Architecture News, 2012, 40(3): 500-511

[3] First the tick, now the tock: Next generation Intel microarchitecture (Nehalem) [OL]. http://www.bitpipe.com/detail/RES/123871608_708.html

[4] Rabl T, Sadoghi M, Jacobsen H A, et al. Solving Big Data Challenges for Enterprise Application Performance Management[J]. PVLDB, 2012, 5(12): 1724-1735

[5] DeCandia G, Hastorun D, Jampani M, et al. Dynamo: Amazon's Highly Available Key-Value Store[J]. ACM Sigops Oper. Syst. rev, 2007, 41(6): 205-220

[6] Cartell R. Scalable SQL and NoSQL data stores[J]. ACM Sigmod Record, 2010, 39(4): 12-27

[7] Nguyen T T, Nguyen M H. Zing Database: high-performance key-value store for large-scale storage service[J]. Vietnam Journal of Computer Science, 2015, 2(1): 13-23

[8] The Apache Cassandra Project[OL]. http://cassandra.apache.org

[9] Chen C, Hsiao M. Bigtable: A distributed storage system for structured data[J]. Proceedings of Osdi, 2006, 26(2): 205-218

[10] Cooper B F, Silberstein A, Tam E, et al. Benchmarking cloud serving systems with YCSB[C]// SoCC. 2010: 143-154

[11] Bridges J T, Dieffenderfer J N, Sartorius T, et al. Caching memory attribute indicators with cached memory data field[P]. US, US20070094475 A1, 2005

[12] Spillane R P, Shetty P J, Zadok E, et al. An efficient multi-tier tablet server storage architecture [C] // Acm Symposium on Cloud Computing Acm. 2011: 1-14

(上接第 170 页)

[11] Li Y, Yang M, Kan H B. Constructing and counting Boolean functions on even variables with maximum algebraic immunity [J]. IEICE Transactions on Fundamentals, 2010, 93-A(3): 640-643

[12] Rizomiliotis P. On the resistance of Boolean functions against algebraic attacks using univariate polynomial representation[J]. IEEE Transactions on Information Theory, 2010, 56(8): 4014-4024

[13] Wu B F, Lin D D. Constructing Boolean Functions with Good Cryptographic Properties by Concatenation[J]. Journal of Cryptologic Research, 2014, 1(1): 64-71(in Chinese)
吴保峰，林东岱. 具有良好密码学性质的布尔函数的级联构造[J]. 密码学报，2014，1(1)：64-71

[14] Wang Q, Peng J, Kan H, et al. Constructions of cryptographically significant Boolean functions using primitive polynomials[J]. IEEE Transactions on Information Theory, 2010, 56(6): 3048-3053

[15] Zhou Q F, Li X X, Qian H F. Construction of almost perfect algebraic immune resilient functions on even variables[J]. Computer Engineering, 2014, 40(12): 74-77(in Chinese)
周祁丰，李祥学，钱海峰. 具有几乎完美代数免疫的偶数元弹性函数构造[J]. 计算机工程，2014，40(12)：74-77

[16] Carlet C. On the higher order nonlinearities of algebraic immune functions[M] // Advances in Cryptology — CRYPTO 2006. Springer Berlin Heidelberg, 2006: 584-601

[17] Carlet C, Feng K. An infinite class of balanced vectorial Boolean functions with optimum algebraic immunity and good nonlinearity[M] // Coding and Cryptology. Springer Berlin Heidelberg, 2009: 1-11

[18] Feng K, Yang J. Vectorial Boolean functions with good cryptographic properties[J]. International Journal of Foundations of Computer Science, 2011, 22(6): 1271-1282

[19] Dong D, Qu L, Fu S, et al. New constructions of vectorial Boolean functions with good cryptographic properties[J]. International Journal of Foundations of Computer Science, 2012, 23(3): 749-760

[20] Lou Y, Han H, Tang C, et al. Constructing vectorial Boolean functions with high algebraic immunity based on group decomposition[J]. International Journal of Computer Mathematics, 2014, 92(3): 451-462

[21] 温巧燕，钮心忻，杨义先. 现代密码学中的布尔函数[M]. 北京：科学出版社，2000

[22] Li C L, Zhang H G, Zeng X Y, et al. The lower bound on the second-order nonlinearity for a class of Bent functions[J]. Chinese Journal of Computers, 2012, 35(8): 1588-1593(in Chinese)
李春雷，张焕国，曾祥勇，等. 一类 Bent 函数的二阶非线性度下界[J]. 计算机学报，2012，35(8)：1588-1593

[23] Sun G H, Wu C K. On the nonlinearity, algebraic degree and algebraic immunity of some symmetric Boolean functions[J]. Chinese Journal of Computers, 2014, 37(11): 2247-2255(in Chinese)
孙光洪，武传坤. 几类对称布尔函数的非线性度、代数次数和代数免疫阶[J]. 计算机学报，2014，37(11)：2247-2255

[24] Zhou Y. Characterization of a Balanced Boolean Function with the Minimum of the Sum-of-squares Indicator[J]. Journal of Cryptologic Research, 2015, 2(1): 17-26(in Chinese)
周宇. 具有最小平方和指标的平衡布尔函数性质刻画[J]. 密码学报，2015，2(1)：17-26

[25] Li W, Wang Z, Huang J. The e-derivative of boolean functions and its application in the fault detection and cryptographic system[J]. Kybernetes, 2011, 40(5/6): 905-911

[26] Huang J L, Wang Z. The relationship between correlation immune and weight of H Boolean function[J]. Journal on Communications, 2012, 33(2): 110-118(in Chinese)
黄景廉，王卓. H 布尔函数的相关免疫性与重量的关系[J]. 通信学报，2012，33(2)：110-118

[27] Zhao M L. Method of detecting special logic function based on Boolean e-derivative[J]. Journal of Zhejiang University (Science Edition), 2014, 41(4): 424-426(in Chinese)
赵美玲. 基于布尔 e 导数的特殊逻辑函数检测方法[J]. 浙江大学学报(理学版)，2014，41(4)：424-426